CHAPTER 6

C

In this chapter

How many jokes begin with a phrase like “A man walks into a bar”? So many that when someone hears that phrase, it is likely that they will assume it is a joke. So, to ruin the joke and speak philosophically, what is a *man*, what is a *bar*, and what does *walking* entail? *Walking* seems to be something that a man can do, an action they can perform. And a *bar* is a place where a *man* can *walk*. Can a *man* do anything else but *walk*? Is a *bar* the only place a man can *walk* to?

It seems silly to examine a sentence in that way, but in the context of a computer program it may be more meaningful. Imagine that this discussion involves a computer game or simulation. A *man* now represents some kind of thing or object that is manipulated by the program. A *man* has properties and things it can do, which is to say operations it can perform. What properties does a *man* have? Well, as a small subset of the possibilities:

Property	Type
Name	String
Sex	Boolean
Phone number	Integer
Height	Float
Weight	Float
Job	String?
Home (location, address)	String?
Interests	Array of String
Income	Float
Possessions (other objects)	Array of object
Spouse	*person*
Children	Array of *person*

So a *man* would appear to be a complex data type having a number of properties. Note especially that a man can have a property or characteristic called **spouse**. A **spouse** is something called a *person*; so is a *man*, really. This is pretty abstract, but stay with it: a *man* is a *person*, and perhaps some of the characteristics of a *man* are really those of (i.e., inherited from) a *person*. In fact, it would appear that most of them are. The only thing that distinguishes a *man* from other persons would (from the list above) be *sex*, which would be (perhaps) **false** for a man and **true** for a *woman*, another kind of *person*.

Imagine that there is a whole class of things called *person* that have most of these properties. A *man* could be derived from this, since *man* has many of these properties in common. A *woman* could be another class, perhaps having a few different properties. A *man* could have, for example, a "date of last prostate exam" as a property, but a *woman* could not. A *woman* could have a "date of last pap smear," but a *man* could not. At some point, person has many common
characteristics, but *man* has some that *woman* does not and vice versa.

So, considering the original proposition: what is a *bar*? It is clearly something (object) that can hold (contain) a *man*. Perhaps it can contain many *men*. *Women*? Why not? If a *person* has to be either a *man* or a *woman*, then a bar can contain some number of *person*s. A *bar* is a class of objects that can hold or contain some number of *person*s. It would be a container class, one supposes, or a holder of some kind.

So, the phrase "A man walks into a bar" might be expressed as:

aMan.walksInto (aBar)

where aMan is a particular man (a specific instance of a man class) and aBar is a specific instance of a class of objects known as bar. This man has a Name, which is to say that one of the properties that a man has is a Name, and this is really just a variable. Since each individual *man* has a **Name**, there has to be a way of getting at (accessing) each one. It is done through each instance, like so:

```
print (aMan.Name) # Accessing /printing the name.
aMan.Name = "Ted Smith" # Assigning to the name.
```

Using this syntax, the dot (".") is placed after the name of the instance. The syntax "**aMan.Name**" means "look at the variable **aMan**, which is an instance of *man*, for a property called **Name**."

Okay, so what is **walksInto** in the above expression `aMan.walksInto(aBar)`? Considering the syntax just described, it would appear to be a function that was a part of the definition of *man*. It takes one parameter, which is something having the type *bar*.

This may all seem very abstract still, but this way of looking at things seems sensible in that it appears to organize information and provide a clear and formal way to access it and manipulate it. This discussion has been a metaphor for the concept of a *class* and the ideas behind *object orientation*, two key elements of modern programming structures. Python permits the programmer to define classes like the *man* or *bar* objects previously described and to use them to encapsulate variables and functions and create convenient modular constructions.

6.1 CLASSES AND TYPES

A **class**, in the general sense, *is a template for something that involves data and operations (functions)*. An **object** is *an instance of a class, a specific instantiation of the template*. Defining a class in Python involves specifying a class name and a collection of variables and functions that will belong to that class. The man class that has been referred to so far has only a few characteristics that we know about for certain. It does have a function called **walksInto**, as one
example. A first draft of the man class could be as follows:

```
class man:
    def walksInto (aBar):
      # code goes here
```

A function that belongs to a class is generally referred to as a *method*. This terminology likely refers back to a language devised in the 1970s named *Smalltalk*. According to the standard for that language, "*A method consists of a
sequence of expressions. Program execution proceeds by sequentially evaluating the expressions in one or more methods.*" In the above example, **walksInto** is a method; essentially, a method is any function that is part of a class.

Classes can have their own data too, which would be variables that 'belong' to the class in that they exist inside it. Such variables can be used inside the class but can't be seen from outside.

Looking closely at the simple class **man** above, notice that it is actually still a rather abstract thing. In the narrative about a man walking into a bar it was a specific *man*, as indicated by a variable **aMan**. So it would seem that a class is really a description of something, and that examples or instances should be created in order to make use of that description. This is correct. In fact, many individual instances of any class can be created (instantiated) and assigned to variables. To create a new instance of the class **man**, the following syntax could be used:

```
aMan = man()
```

When this is done all of the variables used in the definition of man are allocated. In fact, whenever a new man class is created, a special method that is local to man is called to initialize variables. This method is the *constructor*, and can take parameters that help in the initialization. Creating a man might involve giving him a name, so the instantiation may be:

```
aMan = man("Jim Parker")
```

In this case the constructor accepts a parameter, a string, and probably assigns it to a variable local to the class (**Name**, most likely). The constructor is always named __init__:

```
def __init__ (self, parameter1, parameter2, …):
```

The initial parameter named **self** is a reference to the class being defined. Any variable that is a part of this class is referred to by prefixing the variable name with "self." To make a constructor for **man** that accepted a name, it would look like this:

```
 def __init__ (self, name):
     self.Name = name
```

When a man is created, the statement would be:

```
aMan = man ("Jim Parker")
```

This metaphor has fulfilled its purpose for the moment. There are some exercises concerning it at the end of the chapter, but another more practical example might be better now.

6.1.1 The Python Class – Syntax and Semantics

The *man walks into a bar* example illustrates many aspects of the Python class structure but obviously omits many details, especially formal ones that can be so important to a programmer. A **class** looks like a function in that there is a keyword, **class**, and a name and a colon, followed by an indented region of code. Everything in that indented region "belongs" to the class, and cannot be used from outside without using the class name or the name of a variable that is an instance of the class.

The method **__init__** is used to initialize any variables that belong to the class. Java would call this method a *constructor*, and that's how it will be referenced here too. Any variables that belong to the class must be accessed through either an instance (from outside of the class) or by using the name **self** (from within the class). So, **self.name** would refer to a variable that was defined inside of the class, whereas simply using **name** would refer to a variable local to a method. When **__init__** is called, a set of parameters can be passed and used to initialize variables in the class. If the first parameter is **self**, it means that the method can access class-local variables; otherwise it cannot. Normally self is passed to **__init__** or it can't initialize things. Any variable initialized within **__init__** and prefixed by **self** is a class-local variable. Any method that is passed **self** as a parameter can define a new class-local variable, but it makes sense to initialize all of them in one place if that's possible.

A simple example of a class, initialization, and method is:

```
class person:
    def __init__ (self, name):
      self.name = name

    def introduce (self):
      print ("Hi, my name is ", self.name)

me = person("Jim")
me.introduce()
```

This class has two methods, **__init__()** and **introduce()**. After the class is defined, a variable named **me** is defined and is given a new instance of the **person** class having the name “Jim.” Then this variable is used to access the introduce method, which prints the introduction message “Hi, my name is Jim.” A second instance could be created and assigned to a second variable named **you** using:

```
you = person ("Mike")
```

and the method call

```
you.introduce()
```

would result in the message “Hi, my name is Mike.” Any number of instances can be created, and some have the same name as others—they are still distinct instances.

A new class-local variable can be created by any method. In **introduce()**, for example, a new local named **introductions** can be created simply by assigning a value to it.

```
def introduce (self):
    print ("Hi, my name is ", self.name)
    self.introductions = True
```

This variable is **True** if the method introductions has been called. The main program can access this variable directly. If the main program becomes:

```
me = person("Jim")
me.introduce()
print (me.introductions)
```

then the program will generate the output:

```
Hi, my name is Jim
True
```

This is the essential information needed to define and use a class in Python. A more complex example would be useful in seeing how these features can be used in practice.

6.1.2 A Really Simple Class

A common example of a basic class is a point, a place on a plane specified by x and y coordinates. The beginning of this class is:

```
class point:
    def __init__ (self, x, y):
      self.x = x
      self.y = y
```

This simply represents the data associated with a mathematical point. What more does it need? Well, two points have a distance between them. A distance method could be added to the point:

```
def distance (self, p):
    d = (self.x-p.x)*(self.x-p.x) + (self.y-p.y)*
      (self.y-p.y)
    return sqrt(d)
```

If a traditional function were to be used to compute distance, it would be written similarly but not identically. It would take two points as parameters:

```
def distance (p1, p2):
    d = (p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)* (p1.y-p2.y)
    return sqrt(d)
```

The distance method uses one of the points as a preferred parameter, in a sense. The distance between points p1 and p2 would be calculated as:

`d = p1.distance(p2)` or **`d = p2.distance(p1)`**

using the distance method, but as:

```
d = distance (p1, p2)
```

if the function was used. To a degree the difference is a philosophical one. Is *distance* some property that a point has from another point (the method), or is it something that is a thing that is calculated for two things (the function)? A programmer begins, after a while, to see the methods and data of a class as belonging to the object, and as somehow being properties of it. That's what makes a class a type definition.

Many object-oriented languages offer the concept of *accessor* methods. Some languages do not allow variables that belong to a class to be used directly, or allow specific controls on access to them. The truth is that having the ability to find the value of variables and to modify them is generally a bad idea. If the only place that a class local variable can be modified is within the class, then that limits the places where that can occur, and allows more control over what is possible. Preventing errors in programs is partly a matter of restricting actions to a small region, of knowing exactly what is going on at all times.

Similarly, if some object outside of a class has access to the local variables of that class, then it promotes a dependency on a specific implementation, and one of the advantages of an object-oriented implementation is that the interface to the class is fixed and independent of the way that class is implemented. It may seem obvious that a point object has an x, y position and that those would be real numbers, but the point class is the simplest class, and taking advantage of how a class is coded it not always healthy.

All that an *accessor* method does is return a value important to a user of a class. The x and y positions are variables local to the class, and many would agree that they should have an *accessor* method:

```
def getx (self):
    return self.x
def gety (self):
    return self.y
```

Rewriting the **distance()** method to use accessor methods changes it only slightly:

```
def distance (self, p):
    d = (self.x-p.getx())*(self.x-p.getx()) +
      (self.y-p.gety())* (self.y-p.gety())
    return sqrt(d)
```

Methods called *mutators* or *setters* are used to modify the value of a variable in a class. They may do more than that, such as checking ranges and types, and tracking modifications.

```
def setx (self, x):
      self.x = x
    def sety (self, y):
      self.y = y
```

There are other methods that could be added to even this simple class just in case they were needed, such as to draw the point, to return a string that describes the object, to rotate about the origin or some other point, to use a *destructor* method that is called when the object is no longer needed, and so on. Until it is known what the class will be used for, there may not be any value for this effort, but if a class is being provided for general utility, like the Python *string*, as much functionality would be provided as the programmer's imagination could invent. A draw method could simply print the coordinates, and could be useful for
debugging:

```
    def draw (self):
        print ("(", self.x, ",", self.y, ") ")
```

Using this class involves creating instances and using the provided methods, and that should be all. A triangle consists of three points. A triangle *class* could be defined as:

```
  class triangle:
      def __init__ (self, p0, p1, p2):
        self.v0 = p0
        self.v1 = p1
     self.v2 = p2
        self.x = (p0.getx()+p1.getx()+p2.getx())/3
    self.y = (p0.gety()+p1.gety()+p2.gety())/3

      def set_vertices (self, p0, p1, p2):
        self.v0 = p0
        self.v1 = p1
     self.v2 = p2

      def get_vertices (self):
     return ( (self.v0, self.v1, self.v2) )

      def getx (self):
        return self.x
      def gety (self):
        return self.y
```

The (x, y) value of a triangle is its center, or the average value of the x and the y coordinates of the vertices. These are the basic methods. A triangle is likely to be drawn somehow, and the next chapter will explain how to do that specifically. However, without knowing the details, a triangle is a set of lines drawn between the vertices and so might be done that way. As it is, using text only, it will print its vertices:

```
   def draw (self):
       print ("Triangle:")
       self.v0.draw()
       self.v1.draw()
       self.v2.draw()
```

The triangle can be moved to a new position. A change in the x and y location specifies the change, and it is done by changing the coordinates of each of the vertices:

```
   def move (self, dx, dy)
       coord = p0.getx()
       p0.setx(coord+dx)
       coord = p0.gety()
       p0.sety(coord+dy)
       coord = p1.getx()
       p1.setx(coord+dx)
       coord = p0.gety()
       p1.sety(coord+dy)
       coord = p2.getx()
       p2.setx(coord+dx)
       coord = p2.gety()
       p2.sety(coord+dy)
       self.x = self.x + dx
       self.y = self.y + dy
```

In this way of expressing things, it is clear that moving the triangle is a matter of changing the coordinates of the vertices. If each point had a **move()** method, then it would be clearer:

moving a triangle is a matter of moving each of the vertices:

```
   def move (self, dx, dy):
       p0.move(dx, dy)
       p1.move(dx, dy)
       p2.move(dx, dy)
       self.x = self.x + dx
       self.y = self.y + dy
```

Which of these two **move()** methods seems the best description of what is happening? The more complex are the classes, the more value there is in making an effort to design them to effectively communicate their behaviors and to make things easier to expand and modify. It is also plain that the **move()** method for a point is simpler than that for a triangle. That fact is invisible from outside the class, and it actually not relevant.

6.1.3 Encapsulation

In the example of the *point* class there is no actual need for an accessor method, because the variables can be accessed from outside the class, in spite of the arguments that have been given for more controlled use of these variables.
A careful programmer would want to ensure the integrity of classes by forcing the variables to remain protected in some way, and Python allows this while not requiring it.

The variables x and y are accessible and modifiable from outside because of how they are named. Any variable name in a class that begins with an underscore character ("_") cannot be modified by code that does not belong to the class. Such a variable is said to be *protected*. A variable name that begins with two underscore characters can't be modified or even examined from outside of the class, and is said to be *private*. All other variables are *public*. This applies to method names too, so the method **__init__()** that is the usually constructor is private.

Rewriting the point class to make the internal variables private would be done like this:

```
  class point:
      def __init__ (self, x, y):
        self.__x = x
        self.__y = y

  def getx (self):
      return self.__x

  def gety (self):
      return self.__y

  det setx (self, x):
    self.__x = x

  def sety (self, y):
    self.__yy = y

  def distance (self, p):
      d = (self.__x-p.getx())*(self.__x-p.getx()) +
         (self.__y-p.gety())* (self.__y-p.gety())
      return sqrt(d)

  def move(self, dx, dy):
   self.__x = self.__x + dx
   self.__y = self.__y + dy

  def draw (self):
       print ("(", self.__x, ",", self.__y, ") ")
```

Now the internal variables x and y can't be modified or even have their values examined unless explicitly allowed by a method.

6.2 CLASSES AND DATA TYPES

Consider an *integer*. How can it be described so that a person who has not used one before can implement something that looks and acts like an *integer*? This is a specific case of the general problem faced when using computers—to describe a problem in enough detail so that a machine can solve it. The definition could start with the idea that *integers* can be used for counting things. They are numbers that have no fractional part, and that have been extended so that they can be positive or negative.

What can be done with them? Integers can be added and subtracted, multiplied and divided. When dividing two *integers* there can be an *integer* remainder left over. They can be displayed in many forms: as base 10 numbers, in any other base, as Roman numerals, and so on. There are other operations on integers, but these are the most commonly used ones.

What has been done here is to define a *type*. Python types, the ones built into the language, include the *integer type*, as well as floating point numbers, strings, and so on. Each is characterized by an underlying implementation, which is often hidden from the programmer, and a set of operations that are defined on things of that type. This is a fair definition of a type in general. A class can be used to implement a type—not one of the types that the language already efficiently provides, but new types that programmers find useful for their purposes. The **man** and **person** classes described earlier can be thought of as types.

When designing programs that use classes, it is likely that the classes represent types, although they may not be completely implemented. The design scheme would be to sketch a high-level solution and observe what components of that solution look and behave like types. Those components can be implemented as classes. The remainder of the solution will have structure imposed on it by virtue of the fact that these other types exist and are defined to be used in specific ways. Types can hide their implementation, for example. The underlying nature of an integer probably does not matter much to a programmer most times, and so can be hidden behind the class boundary. This has the added feature that it encourages portability: if the implementation has to change, the class can be rewritten while providing the same interface to the programmer.

The operations on the type are implemented as methods. The methods can access the internal structure of the class while providing the desired view of the data and ways of manipulating it. If a class named **integer** existed, for example, then **add()**, **subtract()**, and so on would be methods. Then instances of this class could be implemented:

```
a = integer(5)
b = integer(21)
```

and computing a sum would be:

```
c = a.add(b)
```

The underlying representation of an integer is unknown to a user of this class. All that is known is the interface, described as methods. If the interface is well-documented, then that's all a programmer needs to know. In fact, exposing too much of the class to a programmer can compromise it.

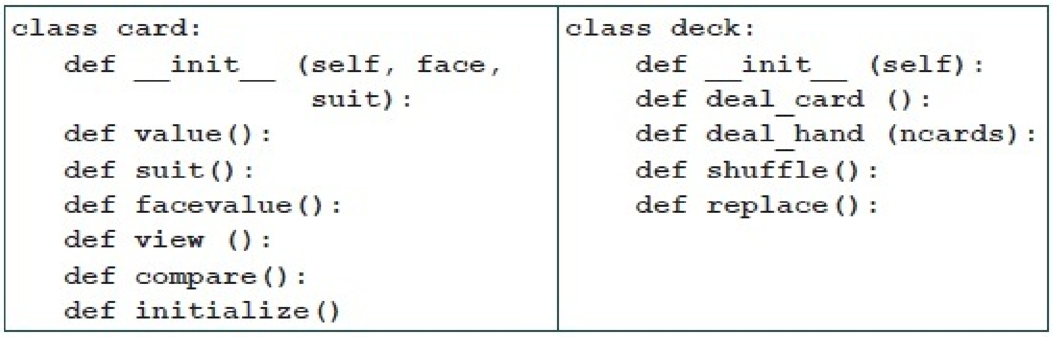
6.2.1 Example: A Deck of Cards

Traditional playing cards these days have red and black colors, four suits, and a total of 52 cards, 13 in each suit. Individual cards are components of a deck, and can be sorted: a 2 is less than a 3, a Jack less than a King, and so on. The Ace is a problem: sometimes it is the high card, sometimes the low card. A card would possess the characteristics of suit and value. When playing card games, cards are dealt from the deck into hands of some number of cards: 13 cards for bridge,
5 for most poker games, and so on. The value of a card usually matters. Sometimes

cards are compared against each other (poker), sometimes the sum of values is key (blackjack, cribbage), and sometimes the suit matters. These uses of a deck of cards can be used to define how classes will be created to implement card games on a computer.

Operations on a card could include to *view* it (it could be face up or face down) and to *compare* it against another card. Comparison operations could include a set of complex specifications to allow for aces being high or low and for some cards having special values (spades, baccarat) so a definition step might be very important.

A deck is a collection of cards. There are usually one of each card in a deck, but in some places (e.g., Las Vegas) there could be four or more complete decks used when playing blackjack. Operations on a deck would include to *shuffle*, to *replace* the entire deck, and to *deal* a card or a hand. With these things in mind, a draft of some Python classes for implementing a card deck can be created:



```
class card:
   def __init__ (self, face,
                 suit):
   def value():
   def suit():
   def facevalue():
   def view ():
   def compare():
   def initialize()
```

```
class deck:
    def __init__ (self):
    def deal_card ():
    def deal_hand (ncards):
    def shuffle():
    def replace():
```

The way that the methods are implemented depends on the underlying representation. When the programmer calls **deal()** they expect the method to return a **card**, which is an instance of the card class. How that happens is not relevant to them, but it is relevant to the person who implements the class. In addition, how it happens may be different on different computers, and as long as the result is the same it does not matter.

For example, a card could be a constant value **r** that represented one of the 52 cards in the deck. The class could contain a set of values for these cards and provide them to programmers as a reference:

```
class card:
    CLUBS_1 = 1
    DIAMONDS_1 = 2
       . . .
    HEARTS_ACE = 51
    SPADES_ACE = 52

    Def __init__ (self, face, suit):
       . . .
```

The variables CLUBS_1, DIAMONDS_1, and so on are accessible in all instances of the card class and have the appropriate value. Variables defined in this way have one instance only, and are shared by all instances.

A second implementation could be as a tuple. The ace of clubs would be (Clubs, 1), for instance. Each has advantages, but these will not be apparent to the user of the class. For example, the tuple implementation makes it easier to determine the suit of a card. This matters to games that have trump suits. The integer value implementation makes it easier to determine values and do specific comparisons. The value of a card could be stored in a tuple named **ranks**, for example, and **ranks[r]** would be a numerical value associated with the specific card.

6.2.2 A Bouncing Ball

Animations and computer simulations see the world as a set of samples captured at discrete times. An animation, for example, is a set of images of some scene taken at fixed time intervals, generally 1/24th of a second or 1/30th of a second. Simulations use time intervals that are appropriate to the thing being simulated. This example is a simulation and animation of a bouncing ball, first in one dimension and then in two dimensions.

A ball dropped from a height **h** falls to the ground when released. Its speed increases as it falls, because it is being pulled downwards by gravity. The basic equation governing its movement is:

$$\mathbf{s = 1/2at^2 + v_0t} \qquad \mathbf{(6.1)}$$

where **s** is the distance fallen at time **t**, $\mathbf{v_0}$ is the velocity the object had at time **t=0**, and **a** is the value of the acceleration. For an object at the earth's surface, the value of **a** is 32 feet/second^2 = 9.8 meters/second^2. For a ball being dropped, $\mathbf{v_0}$ is 0, since it is stationary initially. So, the distances at successive time intervals of 0.5 seconds would be:

Time	S (feet) = 16*t*t	S (meters) = 4.9*t*t
0	0	0
0.5	4	1.225
1	16	4.9
1.5	36	11.025
2	64	19.6
2.5	100	30.625
3	144	44.1

A *class* could be made that would represent a ball. It would have a position and a speed at any given time, and could even be drawn on a computer screen. Making it bounce would be a matter of giving the ball a value that indicated how much of its energy would be lost each time it bounced, meaning that it would eventually stop moving. Writing the code for the class **Ball** could begin with the initialization (the *constructor*):

```
class Ball:
    def __init__(self, height, elasticity):
      self.height = height
      self.e = e
      self.speed = 0.0
      self.a = 32.0
```

This creates and initializes four variables named **height**, **e**, **a**, and **speed** that are local to the class. Remember, the parameter **self** *refers to the class itself*, and any variable that begins with 'self.' is a part of the class. A variable within the function **__init__** that did not begin with 'self.' and was not **global** would belong to the function, and would be created and destroyed each time that function was called.

A method (function) that calculates the height of the ball at a specific time is something else that the **Ball** class should provide. This is simply the value of the class local variable height, so:

```
def height(self):
    return self.height
```

The **self** parameter has to be passed, otherwise the function can't access the local variable **height**. The simulation will need values of height as a function of time, and time will increase in discrete chunks. This could be implemented in a couple of ways: the class could keep track of the time since it was dropped, or it could use the time increment to determine the next speed and position. If the former then a new class variable must be used to store the time; if the latter then a means has to be found to increment the speed rather than using total duration. This second idea is simpler than it sounds. The equation of motion $\mathbf{s = 1/2at^2 + v_0t}$ can use a time increment in place of **t**, and $\mathbf{v_0}$ would be the velocity at the start of the time interval; this yields the new position. The new velocity can be found from a related equation of motion, which is:

$$\mathbf{v = at + v_0} \quad (6.2)$$

where **t** is again the time increment and $\mathbf{v_0}$ is the speed at the beginning of the interval.

The function that updates the speed and position in this manner will be called **delta**:

```
def delta (self, dt):
        s = 0.5*self.a*dt*dt + self.speed*dt
        height = height - s
        self.speed = self.speed + self.a*dt
```

Here the parameter **dt** is the time interval, and so that can be varied by the programmer to get position values at various resolutions.

For now this will be the Ball class. Some code is needed to test this class and show how well (or whether) it works, and this will be the main part of the program. An instance of Ball has to be created and then the delta method will be called repeatedly at time increments of, for an example, 0.1 seconds. A table of height and time can be constructed in this way, and it is a simple matter to see whether the numbers are correct. The main program is:

```
  b = Ball (12.0, 0.5)
  for i in range (0, 20):
      b.delta (0.1)
       print ("At time ", i*0.1, " the ball has fallen to", 
b.height(), " Feet")
```

The results are what should be expected, showing that this class functions correctly:

```
  At time 0.0 the ball has fallen to 12.0 Feet
  ...
  At time 0.5 the ball has fallen to 7.999999999999997 Feet
  ...
  At time 1.0 the ball has fallen to -4.0000000000000036 Feet
  ...
  At time 1.5 the ball has fallen to -24.000000000000004 Feet
  ...
  At time 2.0 the ball has fallen to -52.000000000000014 Feet
  ...
  At time 2.5 the ball has fallen to -88.0000000000003 Feet
  ...
```

Because the initial height was 12 feet, the distance fallen is 12 minus the value given above, so: 4, 16, 36, 64, and 100 feet, which is in agreement with the initial table for the times listed. It appears to work correctly.

This code does not yet do the bounce, though. When the height reaches 0 the ball is at ground level. It should then *bounce*, which is to say begin moving in the reverse direction, with a speed equal to its former downward speed multiplied by the elasticity value. This does not seem hard to do until it is realized that the ball is not likely to reach a height of 0 exactly at a time increment boundary. At one point the ball will be above 0 and then after the next time unit the ball will be below 0. When does it actually hit the ground, and where will be the ball actually be at the end of the time increment? This is not a programming issue so much as an algorithmic or mathematical one, but is a detail that is important to the correctness of the results.

It seems clear that the bounce computation should be performed in the method **delta()**. The height value in the class begins at a positive value and decreases towards 0 as the ball falls. During some specific call to **delta()**, the ball will have a positive height at the beginning of the call and a negative one at the end; this means a bounce should happen. At that time the height of the ball will be negative. The height of the bounced ball at the end of the time interval will be the negated value of the height (i.e., so it is positive again) multiplied by the elasticity.

The speed that should be used in the bounce is based not the final speed but the speed the ball was traveling at the time when the height was 0. This happens when **self.height-s** is zero, or when:

```
self.height = s = 0.5*self.a*dt*dt + self.speed*dt
```

Solve this for the time **xt** that makes the equation work out, which would be the standard solution to a quadratic equation that is taught in high school:

$$xt = \frac{-\text{self.speed} \pm \sqrt{\text{self.speed}^2 + 2a * \text{self.height}}}{a} \tag{6.3}$$

The value of **xt** will be between 0 and **dt**, and is the time within the increment at which the ball struck the ground. At this time the ball will be moving with speed **(self.speed + self.a*xt)** instead of **(self.speed + self.a*dt)** for a normal time interval. The ball will reverse direction and reduce speed by the value of elasticity. Now the ball is moving upwards.

The ball will be slowed by gravity until it stops on its upward path and drops down again. At the top of the path its speed will be 0; at the beginning of the time interval the speed will be negative and at the end it will be positive, and that’s how the peak is detected. This situation is much simpler than the bounce.

The annotated program is as follows:

```
# Ball.py
import math
class Ball:
# Constructor/initializer
    def __init__(self, height, elasticity):
      self.height = height # Current height of the ball
```

```
  self.e = elasticity # How much energy is retained each
bounce
        self.speed = 0.0 # Current speed of the ball,
         # initially 0, down +
        self.a = 32.0 # Acceleration: G= 32 ft/sec^2

  # What Java would call an accessor: not really needed.
      def getHeight(self):
        return self.height

  # Calculate the new height and speed for a change in time of
dt seconds.
      def delta (self, dt):
        startHeight = self.height # Remember the state
         # before dt
        startSpeed = self.speed
        s = 0.5*self.a*dt*dt + self.speed*dt # Equation 1:
         # position update
        self.height = self.height - s
           self.speed = self.speed + self.a*dt # Equation 2:
Speed
         # update
        if self.height < 0: # The sign changed; bounce, when?

       # Equation 3: Solve the quadratic equation to find the
time
      # of bounce
           xt = (-startSpeed - math.sqrt(startSpeed*startSpeed
+2*self.a*startHeight))/self.a
         if xt < 0:
           xt = (-startSpeed + math.sqrt(startSpeed*startSpeed
+2*self.a*startHeight))/self.a
         print ("Bounces at time ", xt)

  # Equation 2 with elasticity
```

```
            self.speed = -(self.speed + self.a*xt)*self.e
            self.height = -self.height * self.e # Correct the
            # height
            if self.e <0.03: self.e = 0.0
    else: self.e = self.e - 0.03

    # Peak of the upward bounce, velocity changes sign from + to
-
    elif startSpeed*self.speed < 0: # If sign differs then
    # the product is -ve
            self.speed = 0 # Speed is 0 at the top
            # of the bounce
            print("Peak")
           print("New speed is ",self.speed," and height starts
at ", self.height)
           if self.height<0.:
            self.height = 0.



    b = Ball (12.0, 0.5) # Initial height 12 feet, elasticity is
0.5
    s = Screen (20, 40)

    for i in range (0, 50):
        b.delta (0.1) # Time increment is 0.1 seconds
```

How can this program be effectively tested? The computed values could be compared against hand calculations, but this is time-consuming. It was done for a few cases and the simulation was accurate. For this example, another program was written in a different programming language to calculate the same values, and the result from the two programs was compared—they were nearly exactly the same. This is not definitive, but is certainly a good indication that this simulation is working properly. In both programs similar approximations were made, and the numbers agreed to seven decimal places.

This class will be expanded later to include an animation of the bouncing ball.

6.2.3 Cat-A-Pult

Early in the development of personal computers, a simple game was created that involved shooting cannons. The player would set an angle and a power level and a cannonball would be fired towards the opposing cannon. If the ball struck the cannon then it would be destroyed, but if not then the opposing player (or the computer) would fire back at the player's cannon. This process would continue until one or the other cannon was destroyed. This game evolved with time, having more complex graphics, mountainous terrain, and more complex aspects. Its influence can be seen in modern games like *Angry Birds*.

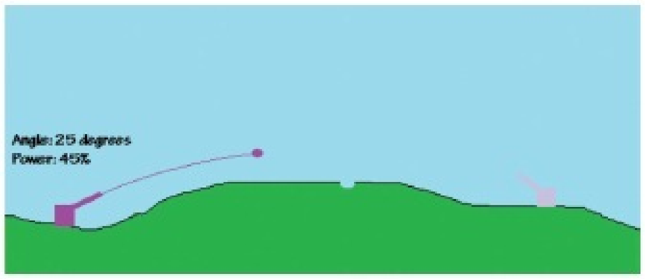


Figure 6.1
Typical configuration of a dueling cannons game.

A variation of this game is proposed as an example of how classes can be used. The basic idea is to eliminate a mouse that is eating your garden by firing cats at it; hence the name **cat-a-pult**. The game will use text as input and output, because no graphics facility is available yet. A player types the angle and the power level and the computer will fire a cat at the mouse. The location where the cat lands will be marked on a simple character display, and the player can try again. The goal is to hit the mouse with as few tries as possible.

Basic Design

Before writing any code, one needs to consider the items in this game and the actions they can take. The items will be *classes*, the actions will be *methods*. There seem to be two items: a *cannonball* (a cat) and a *cannon*. The target (the mouse) could be a class too. The cannon has a location, an angle, and a power or force with which the cannonball will be ejected. Both of the last two factors affect the distance the ball travels. The cannon is given a target as a parameter—in this example the target will be another cannon, basically to avoid making yet another class definition.

The action a cannon can perform is to be *fired*. This involves releasing a cannonball with a particular speed and direction from the location of the cannon. In this implementation an instance of the cannonball class will be created when the cannon is fired and will be given the angle and velocity as initial parameters; the ball will, from then on, be independent. As a class, the ball has a position (x, y) and a speed (dx, dy). The action that it can perform is to move, which will be accomplished using a method named **step()**, and to collide with something, accomplished by the method **testCollision()**.

Detailed Design

In the metaphor of this game, the cannonball is actually a cat and the target is a mouse, but to the program these details are not important. Here's what *is* important:

Class Cannon Class Ball
Has: position x, y position x, y
angle (when fired) speed dx, dy
power (when fired) name (text)
target (another cannon) target (a Cannon class instance)
ball gravity (force changing the height)

Does: fire step
step test for collision

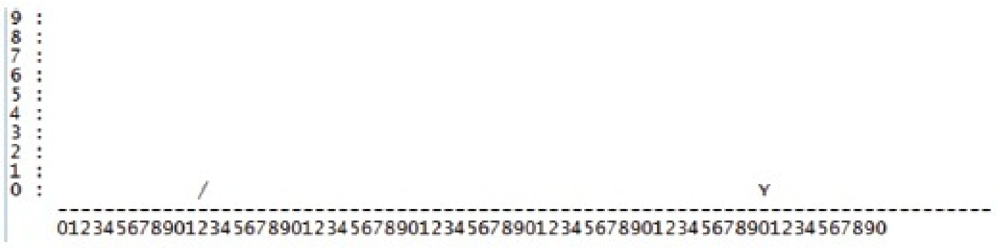
All of the *Has* aspects are class local variables, and in this design they will be initialized within the **__init__** method of each class. This would entail the following:

```
self.x = x  self.x = x
self.y = y  self.y = y
self.power = 0  self.dx = dx
self.angle = 0  self.dy = dy
self.target = target  self.target = target
self.ball = None  self.gravity = 1.0
  self.name = ""
```

The game is essentially one-dimensional. The cannonball will land at a specific x coordinate, and if that is near enough to the x coordinate of the target, then the target is destroyed and the game is over. Without a way to draw proper graphics, this can be imagined as a simple text display with the cannon on one side of the screen and the target on the other, something like that seen in Figure 6.1.

The slash character ("/") on the left represents the cannon, and the "Y" represents the mouse, which is the target. The cannon is at horizontal coordinate 12, and the mouse is at 60; both vertical coordinates are 0.

All of the *Does* aspects represent actions, or things the class object can *do*. When the cannon is fired the ball is created at the cannon coordinates (12, 0) and is given a speed that is related to the angle and power level using the usual trigonometric calculations learned in high school (Figure 6.2):



```
9 :
8 :
7 :
6 :
5 :
4 :
3 :
2 :
1 :
0 :             /                                               Y
    --------------------------------------------------------------------------------
    01234567890123456789012345678901234567890123456789012345678901234567890
```

Figure 6.2
ASCII (text) video of the game at the beginning.

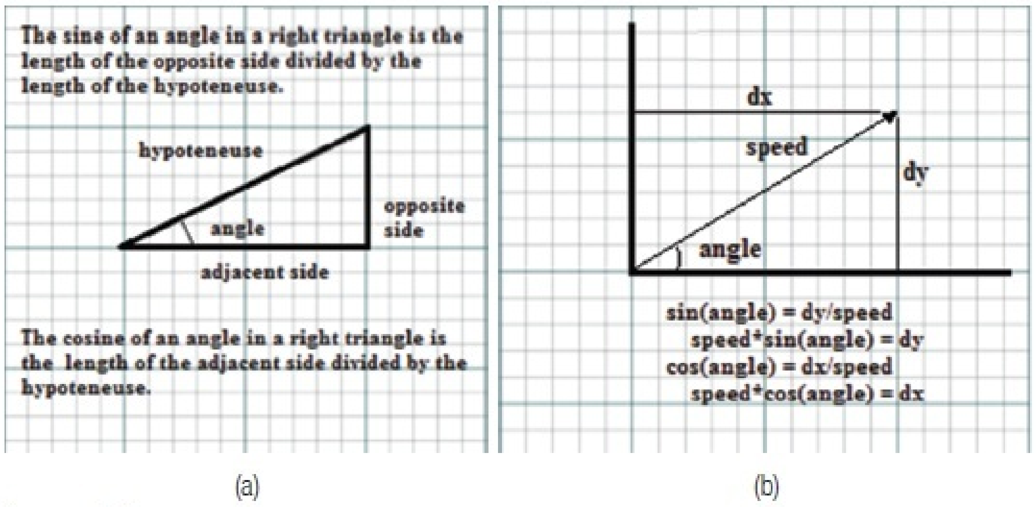

The sine of an angle in a right triangle is the length of the opposite side divided by the length of the hypoteneuse.
hypoteneuse
angle
opposite side
adjacent side
The cosine of an angle in a right triangle is the length of the adjacent side divided by the hypoteneuse.
(a)
dx
speed
dy
angle
sin(angle) = dy/speed
speed*sin(angle) = dy
cos(angle) = dx/speed
speed*cos(angle) = dx
(b)

Figure 6.3
(a) A review of how sines and cosines are computed. (b) using the definition of sine and cosine to calculate the speed of the ball (or any object) in the x and y direction.

```
dy = sin(angle * 3.1415/180.0)
dx = cos(angle * 3.1415/180.0)
```

The angles passed to **sin** and **cos** must be in radians, so the value PI/180 is used to convert degrees into radians. The coordinates in this case have **y** increasing as the ball moves upwards. So, when the cannon is fired a ball is created that has the x and y coordinates of the cannon and the **dx** and **dy** values determined as above. This is accomplished by a method named **fire()**:

```
Fire: takes an angle and a power.
  Angle is in degrees, between 0 and 360
  Power is between 0 and 100 (a percentage)

  1. Compute values for dx and dy from angle and power, where max power is 0.1
  2. Create an instance of Ball giving it x, y, dx, dy, a name ("cat"), and a target (the mouse)
```

The simulation makes time steps of a fixed duration and calculates positions of objects at the end of that step. Each object should have a method that updates the time by one interval, and it will be named **step()**. The cannon does not move, but sometimes has a cannonball that it has fired, so updating the status of the cannon should update the status of the ball as well:

```
Step: make one-time step for this object in the simulation. No parameter.

  1. If a ball has been fired, then update its position. This is done by calling the step()
     method of the ball.
This defines the cannon.
```

The ball must also possess a **step**() method, and it will update the ball's position based on its current speed and location. The **x** position is increased by **dx**, and the **y** is increased by **dy**.

Gravity pulls down on the ball, effectively decreasing the vertical speed of the ball during each interval. After some trials it was determined that the value of **dy** should be decreased by the value of **gravity** during each interval. If the ball strikes the ground, it should stop moving. When does this happen? When **y** becomes smaller than 0. When this occurs, set **dx** and **dy** to 0, and check to see if the impact location is near to the target.

Step: make a one-time step for this object in the simulation. No parameter.

1. Let x = x + dx, changing the x position
2. Let y = y + dy, changing the y position
3. Decrease dy by gravity (dy = dy - gravity)
4. If the ball has struck the ground
5. Let dx = dy = gravity = 0
6. Check for collision with target

Checking to see if the ball hit the target is a matter of looking at the x value of the ball and the x value of the target. If the difference is smaller than some predefined value, say 1.0, then the target was hit. This is determined by a method that will be called **testCollision()**. If the collision occurred then success has been achieved by the player, so set a flag that will end the game.

testCollision: check to see if the ball has hit the target and, if so, set a flag:

1. Subtract the x position of the ball from the x position of the target. Call this **d**.
2. If **d** <= 1.0 then set a flag **done** to **True**.

This defines the class **Ball** and completes the two major classes.

The main program that uses these classes could look something like this:

```
mouse = Cannon (60, 0, None) # Create the target
player = Cannon (12, 0, mouse) # create the cannon
player.fire (42, 65) # Example: fire cannon at
       # 42 degrees 65% power
done = False # initialize variable
       # 'done'
while not done: # so long as the simulation
       # is not over
    player.step() # Update the position of
       # the ball.
```

Actual code for most of this example is shown in Figure 6.4, and the entire program is on the accompanying disc. Included in the disc version is an extra class that draws each state of the game as character graphics that can be displayed in the Python output window; the example in the figure does not include any output, and is unsatisfying to execute. The program on the disc will generate a numeric and graphical representation of the state, showing the axes, the cannon, the ball, and the target after each step. These can be made into distinct text files and can be made into an animation using *MovieMaker* on a Windows computer or *Final Cut* on a Mac. Such an animation is also included on the disc, and is named *catapult.mp4*.

The process by which Cat-a-pult was designed and coded loosely defines a way to design and code almost any program that uses classes.

```
from math import *
class Ball:
    def _ _init_ _ (self, x,
                     y, dx, dy,
                     name,
                     other):
      self.xPos = x
      self.yPos = y
      self.xSpeed = dx
      self.ySpeed = dy
      self.gravity = 1.0
        self.name = name
        self.other = other

    def step (self): # One time
                    # step
      self.xPos = self.xPos +
                  self.xSpeed
      self.yPos = self.yPos +
                  self.ySpeed
      self.ySpeed = self.ySpeed
                    - self.
                    gravity
      if self.yPos < 0:
            self.xSpeed = 0
            self.xSpeed = 0
            self.gravity = 0
            self.yPos = 0
            self.testCollision()
```

```
    def testCollision (self):
        global done
        d = self.xPos-self.
                            other.x
        if d<0: d = -d
        if d < 1.0:
            done = True

class Cannon:
    def __init__ (self, x, y,
                 other):
      self.x = x
      self.y = y
      self.other = other
      self.ball = None

    def fire (self, angle,
               power):
      dy = sin(angle *
                    3.1415/180.0)
      dx = cos(angle *
                    3.1415/180.0)
      self.ball = Ball(self.x,
                      self.y,
                     dx*power/10.0,
                     dy*power/10.0,
                  "Cat", self.other)

    def step (self):
        if self.ball != None:
            (self.ball).step()
```

Figure 6.4
The **Ball** and the **Cannon** classes from the Cat-a-pult simulation

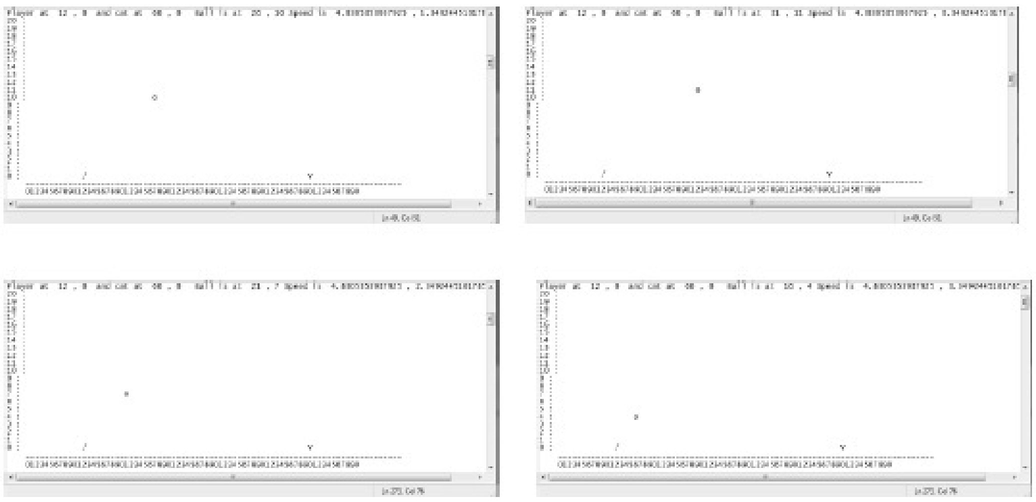


Figure 6.5
Frames from the text animation of the game.

6.3 SUBCLASSES AND INHERITANCE

Classes are designed as language features that can represent a hierarchy of information or structure. A class can be used to define another, and properties from the first class will be passed on (inherited) by the other. A class that is based on another in this way is called a *subclass*, and explanatory examples suffuse the Internet: a pet class with dogs and cats as special cases; a polygon having triangles and rectangles as subclasses; a *dessert* class, having subclasses *pie*, *cake*, and *cookie*; even the initial example in this chapter of a man and a woman class and the person class that they can be derived from. A *subclass* is a more specific case of the *superclass* (or *parent* class) on which it is based.

The examples above are for explanation, and are not really useful as software components, which begs a question about whether subclasses are really useful things. They are, but it requires non-trivial examples to really demonstrate this.

6.3.1 Non-Trivial Example: Objects in a Video Game

To some degree all objects in a game have some things in common. They are things that can interact with other game objects; they have a position within the volume of space defined by the game, and they have a visual appearance. Thus, a description of a class that could implement a game object would include:

```
class gobject:
    position = (0, 0, 0) # Object position in 3D
    visual = None # Graphics that represent
       # the object
    def __init__ (self, pos, vis)
    def getPosition (self):
    def setPosition(self, p):
    def setVisual(self, v):
    def draw (self):
```

Anyone who has played a video game knows that some of the objects can move while others cannot. Objects that move can have their position change, and the position has to be updated regularly. An object that can move can have a speed and a method that updates their position; otherwise it is like a **gobject**. This is a good case for a subclass:

```
class mobject (gobject):
    speed = (0, 0, 0) # Speed in pixels per frame the
       # x,y,z directions
    def __init__ (self, s)
    def getSpeed(self):
    def setSpeed(self, s):
    def move(self):
    def collision(self, gobject):
```

The syntax of this has the superclass **gobject** as a parameter (apparently) of the subclass **mobject** being defined. If an instance of a **gobject** is created, its **__init__** method is called and the resulting reference has access to all of the methods in the **gobject** definition, just as one would expect. If an instance of **mobject** is created, the **__init__** method of **mobject** is called, but not that of **gobject**. Nonetheless, all properties and methods of both classes are available through the **mobject** reference; that is, the following is legal:

```
m = mobject ( (12, 0, 0))  # Create mboject with speed
       # (12,0,0)
m.draw()  # Draw this object
```

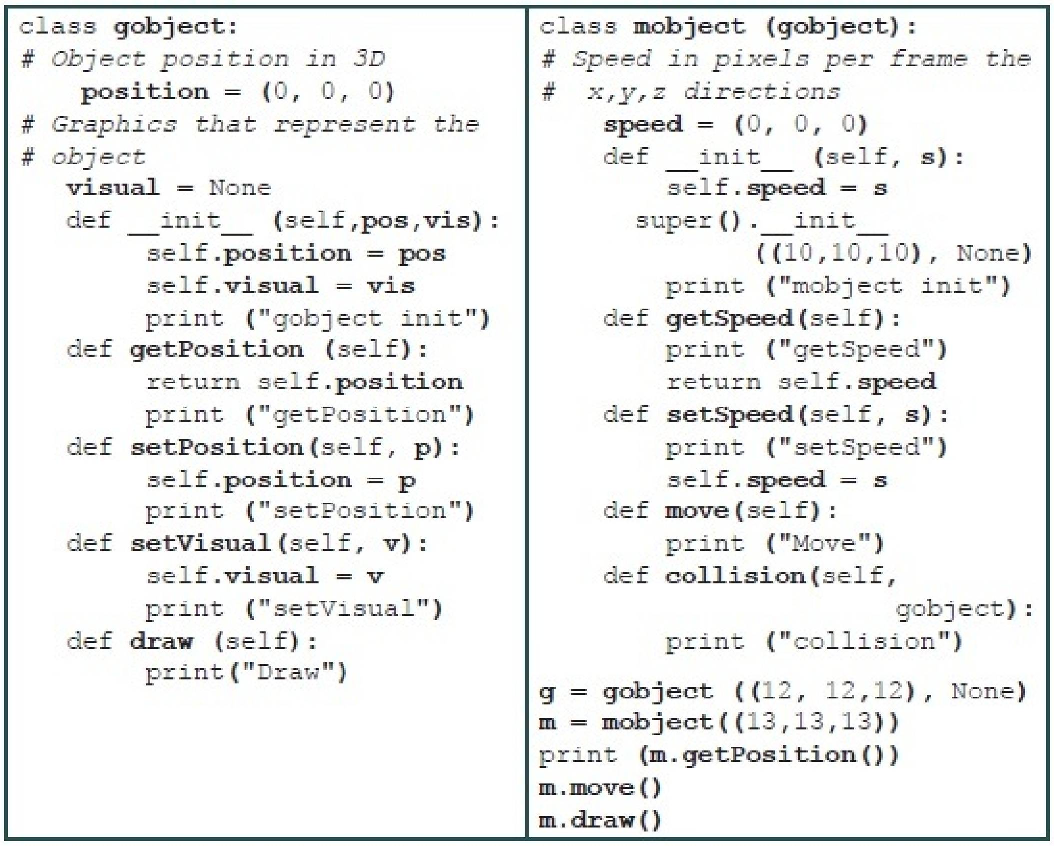
even though an **mobject** does not possess a method **draw()**; the method defined in the parent class is accessible and will be used. When the **mobject** is created it is also a **gobject**, and all of the variables and methods belonging to a **gobject** are defined also. However, the **__init__()** method for **gobject** is not called unless the **mobject** **__init__()** method does so. This means that, for the **mobject**, the values of **position** and **visual** are not specified by the constructor and will take the default values they were given in the **gobject** class. If no such value was given, they will be undefined and an error will occur if they are referenced.

Calling the **__init__()** method of the parent class can be done as follows:

```
super().__init__((10,10,10), None)
```

In this instance the constructor for **gobject** is called, passing a position and a visual. This would normally be done only in the **__init__()** of the subclass.

Now consider the following code. The methods are mainly stubs that print a message, but the output of the program is instructive:



```
class gobject:
# Object position in 3D
    position = (0, 0, 0)
# Graphics that represent the
# object
   visual = None
   def __init__ (self,pos,vis):
        self.position = pos
        self.visual = vis
        print ("gobject init")
   def getPosition (self):
        return self.position
        print ("getPosition")
   def setPosition(self, p):
        self.position = p
        print ("setPosition")
   def setVisual(self, v):
        self.visual = v
        print ("setVisual")
   def draw (self):
        print("Draw")
```

```
class mobject (gobject):
# Speed in pixels per frame the
#  x,y,z directions
    speed = (0, 0, 0)
    def __init__ (self, s):
        self.speed = s
      super().__init__
               ((10,10,10), None)
        print ("mobject init")
    def getSpeed(self):
        print ("getSpeed")
        return self.speed
    def setSpeed(self, s):
        print ("setSpeed")
        self.speed = s
    def move(self):
        print ("Move")
    def collision(self,
                         gobject):
        print ("collision")

g = gobject ((12, 12,12), None)
m = mobject((13,13,13))
print (m.getPosition())
m.move()
m.draw()
```

Output from this is:

```
gobject init  from the creation of the gobject instance g
gobject init  when m is created it calls the parent __init__
mobject init  from the mobject __init__ when m is created
(10, 10, 10)  m.getPosition, showing access to parent
methods
Move  m.move call
Draw  m.draw call, again showing access to parent method
```

Attempting to call **g.move()** would fail because there is no **move()** method within the **gobject** class. Hence, if an object was passed to a function that would attempt to move it, it would be critical to know whether the parameter passed was a **gobject** or an **mobject**. Consider a method that moves an object **x** out of the path of an **mobject** instance if it can, or changes the path of the **mobject** if it cannot. This method, named **dodge()**, might do the following:

```
def dodge self, (x):
    c = x.getPosition()
    c = c + (dx, dy, 0)
    x.setPosition (c)
```

However, if the parameter is an instance of a **gobject**, then it should not be moved. The function **isinstance()** can be used to determine this. The result of:

```
isinstance (x, gobject)
```

will be **True** if x is a **gobject** and **False** otherwise. If **False**, then it can't be moved and the **dodge()** method will have to move the current **mobject** out of the way instead:

```
def dodge self, (x):
    if isinstance(x, gobject):
      self.position = self.position + (dx, dy, 0)
else:
    c = x.getPosition()
    c = c + (dx, dy, 0)
    x.setPosition (c)
```

6.4 DUCK TYPING

In many programming languages, types are immutable and compatibility is enforced. This is not generally true in Python, but still there are operations that require specific types. Indexing into a string or tuple must be done using something much like an integer, and not by using a float. Now that classes can be used to build what amounts to new types, more attention should be paid to the things a type should offer and the requirements this puts on a programmer. A Python philosophy could be that the fewer restrictions the better, and this is a principle of *duck typing* as well.

It should not really matter what the exact type of the object is that is being manipulated, only that it possesses the properties that are needed. In a very simple case, consider the classes point and triangle that were discussed at the beginning of this chapter. It was proposed that both could have a **draw()** method that would create a graphical representation of these on the screen, and both have a **move()** method as well. A function could be written that would move a *triangle* away from a *point* and draw them both:

```
def moveaway (a, b)
    dx = a.getx()-b.getx()
    dy = a.gety()-d.gety()
    a.move (dx/10, dy/10)
    b.move (-dx/10, -dy/10)
```

Question: which of the parameters, **a** or **b**, is the *triangle*, and which is the *point*? Answer: it does not matter. Both classes have the methods needed by this function, namely **getx()**,

gety(), and **move()**. Because of this the calls are symmetrical, and both of the following are the same:

```
 moveaway (a, b)
   moveaway (b, a)
```

In fact, a class that possesses these three methods can be passed to **moveaway()** and a result will be calculated without error. The essence of duck typing is that, so long as an object offers the service needed (i.e., a method of the correct name and parameter set) to another function or method, then the call is acceptable. There is a way to tell whether the class instance **a** has a **getx()** method. The built-in function **hasattr()**:

```
if hasattr (v1, "getx"):
     x = v1.getx()
```

The first argument is a class instance, and the second is the name of the method that is needed, as a string. It returns **True** if the method exists.

The name comes from the old saying that "if something walks like a duck and quacks like a duck, then it *is* a duck." As long as a class offers the things asked for, then it can be used in that context.

6.5 SUMMARY

A **class**, in the general sense, is a template for something that involves data and operations (functions). An **object** is an instance of a class, a specific instantiation of the template. Defining a class in Python involves specifying a class name and a collection of variables and functions that will belong to that class. A **method** is a function that belongs to a class, and so can have easy access to its internal data. As a first parameter, a method can be passed the **self** variable by default, which can be thought of as a reference to the object currently executing. Thus, within a method, the expression **self.x** refers to a variable **x** defined in the class. An object is created using the name of the class: for a class named **thing**, an instance **x** is created using **x = thing()**. When this occurs, if there is a method in **thing** named **__init__**, then that method is called. This is referred to as an initializer or a constructor.

Accessing methods in an object is done using a "dot" notation: **obj.method()**. Variables can be accessed in this way too.

A *subclass* is a class that possesses all of the properties of some other class, the *parent* class or *superclass,* plus some new ones. The data and methods of the parent class can be accessed from the subclass (or *child* class). A subclass of **thing** named **something** would be defined using the syntax:

class something(thing):

A class can represent a new type, where methods represent operations.

Public variables can be accessed and modified from outside of a class; *protected* variables can be accessed but not modified from outside of a class, and must begin with an underscore character (e.g., “_variable”); *private* variables can neither be accessed nor modified from outside of the class, and must begin with two underscore characters (e.g., “__variable”).

The principle of *duck typing* is that it should not really matter what the exact type of the object is that is being manipulated, only that it possesses the properties that are needed.

Exercises

1. Define a class named *square* in which the construct takes the length of the side as a parameter. This class should have a method **area()** that computes and returns the area of the square.

2. Define a subclass of *square* named *button* that also has a location, passed as X and Y parameters to the constructor. A button always has a width of 10. The button class has the following methods:

center()	Return the coordinates of the center of the button
label(s)	Set the value of a text label to be drawn to s

3. Create a class *client*. A client is a data-only class that has no methods other than **__init__()**, but that holds data. In this case the client class holds a **name**, a **category** (retail or commercial), a **time** value (integer), and a **service** value (integer). All values are established when the instance is created by passing parameters to **__init__()**. Now create two subclasses of client, one for each category, *retail* and *commercial*.

4. Define a class named fraction that implements fractional numbers. The constructor takes the numerator and denominator as parameters, and the class provides methods to add, multiply, negate (make negative), print, and find the reciprocal of a fraction. Test this class by calculating:

   14/16 * 3/4

   1/2 – 1/4

   Bonus: results are reduced to smallest possible denominator.

5. Given the following class:

   class value:

   def __init__ (self)

   self.val = randrange(0,100)

   and the initialization:

   t = ()

   for i in range(0,100):

```
v = value()

t = t + (v,)
```

write the code that scans the tuple **t** and locates the smallest integer saved in any of the class instances.

6. Create a class that simulates a NAND logic gate with three inputs. The output will be 1 unless all three inputs are 1, in which case the output is 0. Every time an input is changed, the output is changed to reflect the new state; thus, methods to set each input and to calculate the result will be needed, in addition to a method that returns the output.

Input 1	Input 2	Input 3	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 6.1
Truth table for the 3-input NAND gate.

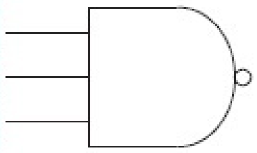


Figure 6.6
The symbolic representation used in a circuit.

7. A queue is a data structure that accepts new (incoming) data at one end (the back) and stores it in the order of arrival, giving the data at the front of the queue when requested. It's like a line at a cashier in a store: customers wait for the cashier in order of arrival. Implement a queue as a class; it has operations **into()** and **out()** to add things and remove things from the queue, and **empty()**, which returns **True** if the queue has no data in it. What is added to the queue are objects of a class *client*, as seen in Exercise 6.3 above.

8. Simulation: The gestation period for a rabbit is 28–32 days, and they will breed a week after having a litter. A female rabbit (a doe) will breed for the first time at about 100 days old. Create a class that represents a rabbit and simulate the growth of a rabbit population that starts with three does at day 0. Assume a litter size of between 3 and 8, and that half of the offspring will be male. Increase time by 1 day at a time and answer the question: "How many rabbits will there be after 1 year?" if the initial population is three does and one male (buck).

Notes and Other Resources

Notes on Python Classes:

http://www.jesshamrick.com/2011/05/18/an-introduction-to-classes-and-inheritance-in-python/

August 12, 2015. *http://componentsprogramming.com/using-the-right-terms-method/*

Duck typing in Python: *http://www.voidspace.org.uk/python/articles/duck_typing.shtml*

1. R. Chugh, P. Rondon, and R. Jhala. (2012, January). **Nested refinements: A logic for duck typing**, *ACM SIGPLAN Notices*, *47*(1), 231–244.
2. Ole-Johan Dahl. (2002). **The birth of object orientation: The Simula languages**, in *Software Pioneers: Contributions to Software Engineering*, edited by Manfred Broy and Ernst Denert, Programming, Software Engineering, and Operating Systems Series, Springer, 79–80.
3. O.-J. Dahl and K. Nygaard. (1968). **Class and subclass declarations,** in *Simulation Programming Languages*, edited by J. Buxton, Proceedings from the IFIP Working Conference in Oslo, May 1967, North Holland.
4. Adele Goldberg and Alan Kay. **Smalltalk-72 Instruction Manual**, 44.
5. *ANSI Smalltalk Standard v1.9 199712 NCITS X3J20 draft*, Section 3.1, 9.
6. B. Liskov, A Snyder, R. Atkinson, and C. Schaffert. (1977). **Abstraction mechanisms in CLU**, *Communications of the ACM*, *20*(8), 564–576.